Express Mail: EV309868750US
Date of Deposit: October 28, 2003

CONTROL FLOW BASED COMPRESSION OF EXECUTION TRACES

Technical Field

This invention relates to a method and apparatus for executing and tracing a program using a compact representation of instructions and memory references.

Background of the Invention

Memory performance studies often employ address traces generated during an execution of a program, to analyze cache behaviors and their effects on program execution time. Address traces capture the order in which memory locations are accessed during execution; however, these traces typically do not carry any direct information on the control flow in the program. On the other hand, architectural studies use instruction traces, which capture the control flow of a program, but do not contain any address traces. Machine simulators often execute or interpret the instructions to obtain the addresses of locations referenced in the program.

In general, when traces get too large, space becomes a premium for their storage. In addition, if compression and de-compression are done off-line (i.e. producing a compressed trace from a given uncompressed trace and vice versa), the space problem is further accentuated. Furthermore, compressed traces often lose the flexibility to segment the traces so that individual segments can be examined or processed concurrently.

When compression is done on memory traces, they can capture certain repeated sequences of addresses and can fold them into compact representations. But, often, the compression mechanism breaks (That is, the memory trace can not be compressed effectively at these breaks.) when the sequence is interspersed with occasional references outside the recognized pattern. These references may be due to conditionals in the program or to loops whose bodies may have a mixture of strided and non-strided references.

Traditionally, the entire program trace was compressed, making it extremely difficult to relate values in the compressed trace to the structural components (such as blocks of the program) of the program. Thus trace analysis becomes cumbersome.

Summary of the Invention

It is therefore an object of this invention to capture both the control flow as well as memory references of a program during execution.

It is therefore another object of this invention to facilitate the association of generalized events with each point in the control flow.

It is therefore another object of this invention to associate values with an event occurring during execution of a program.

It is therefore another object of this invention to provide an efficient compression mechanism for a program trace.

It is therefore another object of this invention to facilitate the composition of a trace of a program as a collection of segments of desired size, so that each segment of the trace can be accessed and processed independently.

Accordingly, this invention employs a compact representation of the execution trace of a program on a machine for tracing and executing the program.

More specifically, with this invention, values associated with each event are compressed separately, thereby providing improved compression.

More specifically, with this invention control flow is captured as a compressed sequence of values with branch events.

More specifically, one aspect of this invention is a method of generating a trace of a program. This method starts with defining a sequence of events for the program. The sequence of values is then determined for each of the defined events during an execution of the program, and each sequence of values is then compressed to generate a compressed sequence of values for each event. These values are then ordered in accordance with information stored in selected events (such as for example, branch events), where the ordered values correspond to the trace.

Often, when an event generates a regular pattern of values, it presents an opportunity for optimization of the corresponding event. Data prefetching and branch prediction are popular examples of this phenomenon. The representation of the trace greatly influences how easily such patterns can be detected and how architectures can react to them. Our proposed compression and representation renders several such opportunities as described below.

With this invention complete execution information is captured and can be used for simulation experiments for studying architectural and program variations.

With this invention values for repeatedly executed events are captured in a very compact form. This compressed form of event-values acts as a signature for the behavior of the event and can be used for analysis and optimization.

Brief Description of the Drawings

Figure 1 graphically illustrates a block which includes a sequence of instructions, along with events and their associated compressed sequences.

Figure 2 graphically illustrates the strided pattern reduction rules.

Figure 3 graphically illustrates the repeated pattern reduction rules.

Figure 4 graphically illustrates segmentation of compressed values and their decompression to a contiguous sequence of values in the uncompressed trace.

Figure 5 graphically illustrates addresses prefetching using compressed patterns of selected events.

Description of the Invention

Control Flow Representation

The binary code of a program can be statically decomposed into a sequence of blocks 11, where each block is a sequence of instructions 15 and 16 in program order, so that the last instruction in a block is always a branch 16, while all preceding instructions in the block are non-branch instructions 15. During execution, normally control enters at the

beginning of a block, after which all subsequent instructions of that block are executed before control transfers to another block. Each block can be uniquely identified in a program. These blocks are used as the basic units in the control flow representation. Control flow is recorded by capturing the flow between blocks. At times, control may enter at an offset into a block. These situations can be handled by remembering the entry offset into a block.

Each block is associated with a sequence of events such as 12 or 13. Each event has a designated type and the type determines the length of the associated event-value. Events can include, for example, loading and storing into memory, branching, memory allocation, and parallel execution constructs. Typically an event is associated with an instruction of the block. For instance, in order to track memory references, load/store instructions are associated with address-events, where the address referenced is the corresponding event-value. While the event value is not shown in Figure 1, it becomes incorporated into a compressed sequence of values 14. The branch instruction 16 at the end of a block is associated with a branch-event 13, and the corresponding event-value is the identity of the block to which control transfers after the branch is executed (See Figure 1). While the event value for the branch target is not shown in Figure 1, it becomes incorporated into a compressed sequence of values 17.

During execution, each time a block is executed, all of its events occur in the order of the execution. For each block 11 visited during execution, our mechanism maintains the list of event-values 14 for that event in a compressed manner. While a variety of compression mechanisms may be designed, we describe here our favored mechanism. Describe below is how a list of event-values is maintained and compressed. First we describe the basic rules we employ to compress an arbitrary sequence of values.

Compression Rules

Our compression mechanism captures only very simple kinds of repetitions: strided sequences of values and repeated strided sequences. A sequence of values is represented as a sequence of patterns, where each pattern is either a strided pattern or a repeat pattern.

Referring to Figure 2, a strided pattern (sp) (21) is of the form, [(b, k), (d1,n1), (d2,n2), .., (dk,nk)], where b is called the base value, k is the depth of nesting, the di are the strides and ni are the number of times a stride is applied. It stands for the sequence of (n1*n2*..*n2) values produced by the pseudo-code: for(i1=0;i1<n1;i1++)for(i2=0;i2<n2;i2++)..for (ik=0; ik<nk; ik++) print (b+i1*d1+i2*d2+...+ik*dk). A single value, v, is represented by the single pair [(v, 0)]. Referring to figure 3, a repeat pattern (rp) (31) is of the form, {sp1, sp2, ..., spk}*n, where spi are strided patterns and n is the repeat count. It simply represents the specified sequence of strided patterns repeated n times. Patterns are formed according to the following simple replacement rules:

Rule 1: (See Figure 2)
[(b,k),(d1,n1),(d2,n2),..,(dk,nk)], [(a^=b,k),(d1,n1),(d2,n2),..,(dk,nk)]
is replaced by [(b, k+1), (b-a, 2) (d1, n1), (d2, n2)... (dk,nk)]

Rule 2: (See Figure 2)
[(b, k), (d1,n1), (d2,n2), .., (dk,nk)], [(b + d1 * n1, k-1),(d2,n2), .., (dk,nk)]
is replaced by [(b, k), (d1, n1+1), (d2, n2)... (dk,nk)].

Rule 3: (See Figure 3)
sp1, sp2... spk, sp1, sp2... spk
is replaced by [sp1, sp2... spk]*2

Rule 4: (See Figure 3)
{sp1, sp2... spk}*n, sp1, sp2... spa

is replaced by [sp1, sp2... spk]*(n+1)

Compression Mechanism

The trace mechanism maintains the list of blocks executed at any time. The first time, a block is executed, it is added to the list and the value-sequence is initialized to NULL for each of the events of that block. As each value v is obtained for an event, its value-sequence is updated as follows:

1. The strided pattern [(v, 0)] is appended to the right of its value-sequence.
2. Repeatedly reduce the two rightmost patterns of the sequence by rules 1 or 2 when applicable.
3. Search the sequence, from right to left, until either the rightmost pattern is repeated, as in rule 3 or a repeat pattern is found as in rule 4. If none, quit.
4. Reduce by rule 3 or 4, as applicable.

Trace Compression and Decompression

Our preferred storage mechanism for the trace is to store the list of all the blocks encountered in the execution and for each block, the value-sequences of each event of that block are also stored. Given a compressed trace, the decompression is the process of generating the sequence of blocks visited by the execution and for each visit of a block, generating the values corresponding to each event of that block. This can easily be done, by the following algorithm. The algorithm maintains two pointers: current-block and current-event, which are initialized to point to the first block and the first event in that block, respectively. For each strided sequence of the form [(b, k),(d1,n1), (d2,n2), .., (dk,nk)] additional counters, ci, are maintained as shown in [(b, k), (d1,n1,c1), (d2,n2,c2),..,(dk,nk,ck)]. The counters ci are all initialized to zeroes and are used for enumeration. The following steps are repeated until all values are generated:

1. If the head of the value-sequence of the current-event is a strided pattern of the form [(b, k),(d1,n1,c1),(d2,n2,c2),..,(dk,nk,ck)], then compute the event-value v = (b+c1*d1+c2*d2 +..+ck*dk) and advance the counters lexicographically. That is, find the largest 1 <= i <= k such that ci < di-1 and increment ci and reset all cj, j>i to zero. If no such i is found, delete this pattern from the sequence.
2. If the current-event is a branch-event, then reset the current-block to the block identified by v and reset the current-event to its first event.
3. Otherwise generate the value v and advance current-event to the next event in the current-block.

Segmented Traces

In the preceding description, as the trace grows longer, the number of blocks 11 maintained increases and hence searching for them to record the control flow becomes slower. The events and compressed sequences of values (12, 13, 14, and 17) are again shown in Figure 4. Furthermore, during decompression, blocks will have to be read from the trace in a criss-cross manner, depending upon the structure of the code. To alleviate these problems and to facilitate direct processing of selected portions, we describe the following segmentation scheme, (See figure 4). While there can be a variety of criteria for determining segment 41 boundaries, described below is our preferred approach, based on the size of a compressed segment. The idea is to be able to read each segment into memory once and complete all its processing within the memory.

The trace compression algorithm described in the preceding sections is such that a segment can be terminated after any branch instruction (i.e. at the end of any block). As a segment is composed, one can estimate the size of the compressed segment, based on the

number and type of patterns the segment has at any time. If the size exceeds a chosen threshold, the segment is terminated at the end of the current block. This is indicated by specifying the last branch target as a block which does not exist in the current segment. The segment is written out and a new segment starts with empty value-sequences. Referring to Figure 4, when each compressed segment is decompressed, one gets a contiguous sequence of values 43 in the uncompressed trace 42. While one loses the opportunity to compress patterns that spread across segments, the segments offer greater flexibility in handling the trace. An index listing the file offsets to individual segments facilitates seeking to a desired segment and processing it concurrently with processing of other segments.

Hardware Aids for Data Pre-fetching

The preceding discussion focused on generating a compressed trace and processing decompressed traces. Analysis from previous executions of programs can identify events in the control flow that have a well-structured strided reference pattern and software can supply hints to the hardware at appropriate places to initiate data pre-fetches. The same idea can also be used to build automatic data pre-fetch mechanism. Illustrated below is a simple mechanism for pre-fetching strided patterns. Referring to Figure 5, a processor 50 can be equipped with a pre-fetcher 54 that acts on hints supplied by software. Each memory fetch instruction 51 carries its signature (i.e. compressed pattern) 52 with it, when it is compact. The instruction is fed to the execution 53 unit and the signature is fed to the pre-fetcher. When the pre-fetcher receives a signature, it initiates the appropriate pre-fetch command to the memory subsystem. Alternatively the pre-fetcher can also be designed to do the online compression for selected events and trigger the pre-fetching when a stable signature is identified. Similar technique can be employed for the branch target-sequence to trigger branch prediction and hedge fetching.